

AD-A231 316

TECHNICAL REPORT 90-08

①

# Stream Editing for Animation

by

J. K. Kearney and S. Hansen

DTIC  
ELECTE  
JAN 23 1991  
S B D

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF IOWA • IOWA CITY

SECTION 87(2)(b) &

Approved for public release  
Distribution Unlimited

THE UNIVERSITY OF IOWA  
DEPARTMENT OF COMPUTER SCIENCE

List of Technical Reports 1975-

- 75-01\* R. J. Baron and A. Critcher, *STRINGS<sup>2</sup>: Some FORTRAN-callable string processing routines.*
- 75-02 H. Rich, *SPITBOL GROPE: A collection of SPITBOL functions for working with graph and list structures.*
- 75-03 A. C. Fleck, *Formal languages and iterated functions with an application to pattern representations.*
- 75-04 A. Mukhopadhyay, *Two-processor schedules for independent task systems with bounded execution.*
- 75-05\* D. Leinbaugh, *Finite array schemata.*
- 75-06\* T. Sjoerdsma, *An interactive pseudo-assembler and its use in a basic computer science course.*
- 75-07\* J. Lowther, *The non-existence of optimizers and subrecursive languages.*
- 76-01 A. Deb, *Parallel numerical computation.*
- 76-02\* D. Buchrer, *Natural resolution: A human-oriented logic based on the resolution.*
- 76-03\* C. Yang, *A class of hybrid list file organization.*
- 76-04 C. Yang, *Avoid redundant record accesses in unsorted multilist file organizations.*
- 77-01 A. Mukhopadhyay, *A fast algorithm for the longest common subsequence problem.*
- 77-02\* D. Alton and D. Eckstein, *Parallel searching of non-sparse graphs.*
- 78-01 C. Yang and G. Salton, *Best-match querying in general database systems.*
- 78-02 C. Yang and R. Hartman, *Extended semantics to the entity-relationship model.*
- 78-03 A. Mukhopadhyay and A. Hurson, *ASL—an associative search language for data base management and its hardware implementation.*
- 78-04 D. Riley, *The design and applications of a computer architecture utilizing a single control processor and an expandable number of distributed network processors.*
- 78-05 A. Critcher, *Function schemata.*
- 79-01\* D. Willard, *Polygon retrieval.*
- 79-02 S. R. Seidel, *Language recognition and the synchronization of cellular automata.*
- 79-03\* R. J. Baron, *Mechanisms of human facial recognition.*
- 79-04 R. J. Baron, *Information storage in the brain.*
- 80-01 D. E. Willard, *K-d trees in a dynamic environment.*
- 80-02 D. M. Dungan, *Bibliography on data types.*
- 80-03\* D. M. Dungan, *Variations on data type equivalence.*
- 80-04\* R. Baron and S. Zimmerman, *BODIES: A collection of FORTRAN IV procedures for creating and manipulating body representations.*
- 81-01 R. F. Ford, Jr., *Design of abstract structures to facilitate storage structure selection.*
- 81-02 K. V. S. Bhat, *A graph theoretic approach to switching function minimization.*
- 81-03\* K. V. S. Bhat, *On the notion of fuzzy consensus.*
- 81-04\* D. W. Jones, *The systematic design of a protection mechanism to support a high level language.*
- 81-05\* J. T. O'Donnell, *A systolic associative LISP computer architecture with incremental parallel storage management.*
- 81-06 K. V. S. Bhat, *An efficient approach for fault diagnosis in a Boolean n-cube array of microprocessors.*
- 81-07 K. V. S. Bhat, *On "Fault diagnosis in a Boolean n-cube array of microprocessors."*
- 82-01 K. V. S. Bhat, *Algorithms for finding diagnosability level and t-diagnosis in a network of processors.*
- 82-02\* R. K. Shultz, *A performance analysis of database computers.*
- 82-03\* D. W. Jones, *Machine independent SMAL: A symbolic macro assembly language.*
- 82-04 C. T. Haynes, *A theory of data type representation independence.*
- 82-05 P. G. Gyllstrom, *Fault-tolerant synchronization in distributed computer systems.*
- 83-01 C. Denbaum, *A demand-driven, coroutine-based implementation of a nonprocedural language.*
- 83-02 A. C. Fleck, *A proposal for comparison of types in Pascal and associated models.*
- 83-03 S. Yang, *A string pattern matching algorithm for pattern equation systems with reversal.*
- 83-04\* K. W. Miller, *Programming in vision research using pixelspaces, a data abstraction.*
- 83-05 C. Marlin, *A methodical approach to the design of programming languages and its application to the design of a coroutine language.*
- 83-06 K. V. S. Bhat, *An optimum reliable network architecture.*
- 83-07 R. Ford and K. Miller, *An abstract data type development and implementation methodology.*
- 83-08 T. Rus, *TICS system: a compiler generator.*
- 83-09 C. Marlin and D. Freidel, *A model for communication in programming languages with buffered message-passing.*

\* no longer available.

Continued on inside back cover

## Stream Editing for Animation

J. K. Kearney  
S. Hansen

### abstract

The first step in creating a computer animation is often the definition of a time-varying geometric model. Realistic movement sequences can be obtained through physically-based simulation and optimization programs. This paper presents a method for representing movement sequences as streams and describes a system for editing motion streams. The approach allows motion streams to be filtered, duplicated, transformed, and combined. Cameras with prescribed motions can be introduced to visualize scenes. A general facility for composing sequences of spatial transforms permits the specification of complex trajectories and relative motion. Motion stream editing facilitates the synthesis of intricate movement sequences that can be rendered as animation or presented as the input circumstances for additional simulation.



This work was supported in part by National Science Foundation Grant IRI-8808896 and Office of Naval Research Grant ONR 00014-88K-0632.

Statement "A" per telcon Dr. Alan  
Meyrowitz. Office of Naval Research/  
code 1133.

VHG

1/22/91

For	
I	<input checked="checked" type="checkbox"/>
	<input type="checkbox"/>
	<input type="checkbox"/>

per Telcon

Availability Codes

Avail and/or  
Special

Dist

A-1

## 1. Introduction

Three-dimensional motion modeling is playing an increasingly important role in computer animation.<sup>8</sup> Life-like animations can be created by rendering images of time-varying object models. The objects in the scene may have complex shapes, articulations, and may deform over time. Powerful motion generation tools enable designers to create physically realistic and biologically plausible object motions through dynamic simulation and constraint-based optimization.<sup>2-6,9,11</sup> Motion models can also be determined from measurements of physical motions using techniques of cinematography, photogrammetry, and accelerometry. This paper presents a system for combining and modifying motion sequences and for integrating viewing models with object movements.

The state of an object undergoing change can be specified by a model of the object and a sequence of time-varying model parameters. For a rigid body, the model must include a description of object geometry. The motion of a rigid body can be characterized by the time-variation of a coordinate system affixed to the object. Object geometry is most naturally defined relative to the dynamic, object-based coordinate system. Points on the object can be easily mapped into a different coordinate system given the relationship between the object-centered coordinate system and another coordinate system. Following the terminology of robotics, we will call the object-based coordinate system a *frame*. The motion of a rigid body is completely described by a geometric model and a sequence of frames.

Given geometric models and frames for a set of objects, we can construct a scene populated with instances of the modeled objects. The motions of these objects can be represented in a sequence of scenes. Each scene is a snapshot of the time-varying environment. To visualize a scene, a camera model must be defined. If the camera is to move relative to objects in the scene, there must be a means of associating the camera model with changing object frames.

In this paper, we present a system for editing motion sequences based on the use of streams to represent dynamic quantities. These streams can be transformed, merged, and edited after creation to produce rich, dynamic environments. Camera motion is integrated with the motion of volumetric objects. A general facility for constructing spatial transforms permits the specification of complex trajectories and relative motion. The resulting motion sequences can be rendered as animation or presented as the input circumstances for simulation. Man-in-the-loop simulation offers enormous potential for testing and training of human operators in hazardous situations such as flying, driving, teleoperation of robots. The synthesis of realistic situations is critical for the effective development of computer simulated training and testing applications. Streams provide a conceptual framework for composing complex dynamic settings and a computational model that is robust, reliable, and simple to implement.

## 2. An Example of Stream Editing

We begin with an illustration of stream manipulation. The example demonstrates how a single stream acquired from a physically-based simulation of walking can be duplicated, transformed, and assembled into a new stream that represents a precision drill team marching in formation. Our starting point is a stream that models a walking figure. The stream was produced by the physical simulator **newton**.<sup>5,6</sup> The algorithms controlling the walker are described elsewhere.<sup>10</sup> To produce a second walking figure whose path is displaced from the first, we create a transformed copy of the walking stream. The two streams are merged to create a stream with a pair of figures, walking in step with one another. The pair of walking figures can be transformed and merged to obtain a quartet of walkers. This process can be repeated a number of times to produce a whole brigade. The brigade is represented as a single stream that can be reproduced, reoriented, repositioned, and combined with the original marchers to create two divisions marching in different directions. With a judicious selection of the transformation parameters, we can have the two formations cross paths, with members of each brigade passing between members of the other.

A scene from the motion sequence with two groups of four walkers is pictured in Figure 1. The operations required to assemble this model are schematically shown in Figure 2. Edges indicate data streams and boxes indicate stream producing operations. One of our long range goals is to build a graphic programming language similar to these schematic diagrams.

The marching example illustrates the simplicity and power of animation stream editing. To support the operations used in this example, we've designed and implemented a scheme for representing object motion sequences as streams and a set of operations for editing these streams. In this system, the marching example can be constructed even more compactly than presented above by using transform generators that allow sequences of transforms to be defined.

## 3. Streams

A stream is simply a sequence of data elements.<sup>1</sup> The components of the sequence may be arbitrarily complicated and the sequence may be infinitely long. For our purposes, streams will always be homogeneous. That is, the elements in a stream will all be of the same type. The components may be numbers, vectors, functions, or representations of physical entities. Streams are typically conceptualized as a time history of an entity.

To accommodate the possibility of infinitely long sequences, streams incorporate lazy evaluation. A non-empty stream is defined by its first member, the head of the stream, and a promise that when needed, the rest of the stream, called the tail, will be made available.

Three basic stream operations were sufficient to achieve nearly all of the capability required for our system. The first operation is to filter all components satisfying some predicate from a stream. The value returned by a filtering operation is a copy of the stream with filtered elements omitted. For example, given a stream of the natural numbers,  $N = 1, 2, 3, \dots$ , and a predicate,  $(\text{even? } x)$ , that is true whenever a number  $x$  is divisible by 2, we can derive a stream of odd, positive integers by  $(\text{filter } \#'\text{even? } N)$ .<sup>1</sup>

The second basic operation is to map a function over a set of streams. The Lisp form

$(\text{stream-map } \text{func } \text{stream-1 } \text{stream-2 } \dots \text{stream-n})$

defines a stream that is the result of applying the function *func* of  $n$  arguments to successive members of the streams *stream-1*, *stream-2*, ..., *stream-n*. The resulting stream terminates as soon as any one of the stream arguments terminates.

The third stream operation frequently used in the system is to merge two streams together. The result is the stream formed by concatenating one stream after the other. There is the risk that the second stream may be preceded by a infinitely long stream. This creates no technical difficulty. However, if the user's intention is to eventually process both input streams, undesired results may occur.

Stream operations focus attention on the flow of data through computational processes. Successive members of the stream funnel through filters or functions that are mapped over the stream. As a consequence, graph representations such as the one shown in Figure 2 are well suited for visualizing networks of stream operations. This figure schematically shows the stream operations used to create the marching example pictured in Figure 1.

Streams may be assigned names and may be shared by many functions. For example, another way to construct the stream of all even integers is to map addition over two instances of the stream of natural numbers:

$(\text{setq } \text{even } (\text{stream-map } + N N)).$

---

<sup>1</sup>Stream operations will be presented in the syntax of Common Lisp.<sup>7</sup> The prefix operator  $\#'$  identifies the following name as a reference to a function.

where  $N$  is defined as above. This form will create a stream, called *even*, of all even, positive integers by computing  $(1+1, 2+2, 3+3, \dots)$ . In the example above, we can think of there being two copies of the stream  $N$ . However, in fact the mapping function traverses a single copy in parallel. To guarantee that the integrity of a stream is preserved, it is strictly forbidden to change any member of a stream or restructure a stream. Just as we expect a function  $f()$  to always return the same value at  $f(x)$ , we expect the  $i$ th member of a stream to be consistent.

New elements can be created from old ones and at times it is natural to think of the new sequence as a transformed copy of an old sequence. For example, given an object trajectory represented by a stream of positions, *P-stream*, we may want to transform each position such that its motion is relative to another moving object. If the time-varying origin of the reference coordinate system is represented in the stream *R-stream*, then the transformed stream can be expressed as

*(stream-map #'pos-trans R-stream P-stream)*

where *pos-trans* is a function of two positions that returns the second argument transformed with reference to the first. If positions are represented as vectors, then the function *pos-trans* is simply vector addition. It is critical that the transform function return a new position and leave its arguments unchanged.

Data elements may be shared within streams and across streams. For example, the stream produced by filtering all positions above the  $X$ - $Y$  plane will contain precisely the same elements, excluding those above the  $X$ - $Y$  plane, that are in the original stream. Because of the restriction on modifying the contents of a stream, elements or parts of elements can be freely shared with no threat of conflict or inconsistency. Frequent sharing of elements can lead to significant gains in efficiency.

#### 4. Object Representation

The representation of a physical object in the system is a Lisp structure that includes a name, a frame, and a model. Two types of object models are currently represented in the system. A solid object represents the volume occupied by a 3-dimensional body with a geometric model. A camera represents a mathematical model of a camera. The camera model includes a projection function (orthogonal or perspective) and projection parameters such as focal length, scaling, and radial distortion. This information is sufficient to map the scene onto a continuous, two dimensional image plane.

The attachment of a name to an object permits reference critical for filtering and selection operations. Names also provide a means to encode structural information through common naming patterns. For example, a set of objects representing the parts of a body may be labeled "body.arm", "body.torso", etc.

A set of transformation operations return new objects derived from transformations on input objects. A geometric transform rotates and translates a target object. The amount of rotation and translation is determined by the frame of a reference object. The target object's frame is interpreted as being specified in the coordinate system of the reference object. The frame of the new object is the mapping of the target frame into the global coordinate system. This causes the target frame to be rotated and translated exactly as the reference frame is rotated and translated with respect to the global frame. The names and models of new objects are retained from the target objects. Using a geometric transform, an object can be defined to be in a specified relation to another object. This is especially useful for attaching cameras to objects.

Another useful transform is object renaming. When a renaming operation is performed a new object is returned with a model and frame identical to the input object and with a transformed name. The name change may be optionally formed by attaching a prefix or suffix to the name of the input object or by substituting a wholly different name. By prepending the same name to a collection of objects, multiple instances of a compound structure can be distinguished. For example, a set of logically related body parts could be renamed "B1.body.arm", "B1.body.torso", etc.

## 5. Simple streams

Sequences of objects are represented as object streams. Streams of objects are used for two conceptually different purposes. To distinguish these different roles, we will call a stream of objects representing the time history of a single logical object an **object stream** and a stream of objects representing a configuration of objects co-existing at a point in time a **scene stream** or simply a **scene**. The motion of a camera or a solid object may be described with an object stream.

The three basic stream operations described above have important applications with object and scene streams. Two scenes can be combined using the merge operation to form a composite scene consisting of all objects from both input scenes. Two object motion sequences can be spliced together using the same merge operation. The results is a composite stream consisting of the first motion followed by the second motion.

Filtering is useful for deleting objects from a scene. Predicates exist to identify objects with names that match specified sub-strings. Coupled with the name assignment operations described above, this provides a powerful tool for removing object sets. For example, the following form removes all objects with names that contain the string "body" from a scene stream:

*(filter (name-pred "body") scene-stream).*

The function *name-pred* constructs a predicate that takes an object as its single argument and returns **true** if the object's name contains the string passed to *name-pred*. The *filter* function will apply the predicate to successive members of the scene-stream and return a stream containing only those members that did not satisfy the predicate.

A second version of the filter function, *(nfilter pred stream)*, retains only those members of the stream that satisfy its predicate argument. A stream is split into disjoint sets by the combination of *filter* and *nfilter*.

Closely related to filtering, are a collection of functions that extract elements from a stream. The most simple of these is the *(head stream)* function that returns the first member of a stream. Unlike filtering, the result is an element, not a stream. There are functions for selection by index in the stream and to choose the first component satisfying a predicate.

The entire set of objects in a simple stream may be transformed by mapping an object transform operation over the stream. In this way, an object's motion may be tied to another object's motion. Given two object streams, the form

*(stream-map #'geo-trans object-stream1 object-stream2)*

will return a stream of objects derived from *object-stream2* by geometric transforms based on successive members of *object-stream1*.

It frequently occurs that all objects in stream must be transformed by the same reference object. For example, to shift and rotate all objects in a scene as a unit, the same geometric transform must be applied to each member of the stream. The mapping function, however, requires that a stream of reference objects be provided to pass to the transformation operation. Streams provide a simple solution to this problem in the form of an infinite stream of the same object. The function

*(stream-of x)*

returns an infinitely repeating sequence of *x*'s. The infinite stream of objects wastes no space, as only a single copy of the object is explicitly represented. We can think the infinite stream as a non-exhaustive source of the repeated element. The problem of applying a single transform to a scene is solved by

*(stream-map #'geo-trans (stream-of ref-object) scene-stream).*

Every object in the target stream will be transformed by the same reference object.

## 6. Motion Streams: Nested Streams

A dynamic environment is represented as a stream of scenes. We call this nested stream structure a **motion stream**. Conceptually, a motion stream represents the time history of a collection of objects. Each scene within a motion stream represents the instantaneous state of a set of objects.

As with object streams, motion streams can be spliced by merging two streams. Although it possible to filter scenes from a motion stream, the more common application of filtering is to remove objects from the motion sequence. This can be accomplished by mapping a filter over the motion stream. The filter will be applied to each scene stream, deleting the identified objects. To remove all objects containing "bob" as part of their name we use

*(stream-map #'filter (stream-of (name-pred "bob"))) motion-stream).*

This form applies a filter to each scene of the motion-stream. The predicate is supplied by an infinite stream of the "bob" predicate function.

Corresponding scenes of two motion streams may be combined to produce an aggregate stream. The scene-level merger of motion streams is called a confluence. For example, given a motion stream modeling a walking figure, we can create pair of figures walking side-by-side by transforming the first stream and then merging the corresponding scenes of the two streams. To merge two motion-streams, we need only map the stream merge function over the pair of

motion streams.

Whole motion streams can be transformed by applying an object transform function to each object in every scene. To accomplish a transformation of all objects, two levels of mapping must be performed. A useful application of nested mapping is to rename all objects in a motion stream. For example, to prepend the name "*Zambini*" to all objects in the motion stream *trout-stream* we execute

```
(stream-map
  #'stream-map
  (stream-of name-prefix)
  (stream-of (stream-of "Zambini"))
  trout-stream).
```

The outer mapping function passes successive members of three streams to the inner mapping function. The first of these streams is a stream of functions that are to be mapped over the members of the second and third stream arguments. In this example, the same function is supplied until one of the other stream arguments terminates. The second and third stream arguments supply streams of streams to the outer map. The inner mapping function causes the function *name-prefix* to be mapped over each successive scene of the *trout-stream*.

The nested mapping of transforms over motion streams is sufficiently common that a set of these routines have been encapsulated as functions. The renaming function above is simply executed as

```
(mstream-name-prefix "Zambini" trout-stream).
```

Another stream transform allows a motion stream to be repositioned and reoriented. The function

```
(mstream-geo-trans object-stream motion-stream)
```

applies a geometric transform to each scene of *motion-stream* using object's from *object-stream* as referents. The scene transform is accomplished by transforming the component

objects all in the same way.

Frequently, the desired referent object exists as a constituent of a motion stream. In this case, the referent must be selected from each scene of the motion stream to generate a reference object stream. This is accomplished by mapping a selection function over the motion stream.

## 7. Springs: Stream Sources

A set of base streams from which other streams can be derived is needed. One source of stream data is physically-based simulation and motion optimization programs. Tools are also provided for object and motion stream construction. Creating an object or motion stream can be very tedious, and it is not the intention that the system should be used for composition of original motion models. However, there are cases in which simple streams are required that can easily be created. To reposition or reorient a motion stream, for example, a stream of referent objects must be created. For this purpose, the model of the referent object is unimportant. The frame of the referent object is the only attribute of the object that need be defined. A function to create a modelless object with a specified position and orientation can be used to construct the referent object. To apply a constant transform to all scenes in a motion stream, a recursive, one element stream of the referent object can be created with the function *stream-of*. Functions to define cameras and solid objects with simple geometry are also available.

Object generators are provided for creating functionally defined trajectories. Given a function (*func i*) that returns frames along a parametrically defined space curve, the function (*gen-stream func*) will return a stream of objects with frames at (*func 1*), (*func 2*), .... For example, let the function (*simple-path i*) return a frame located at (*i,0,0*) oriented as the global coordinate system. Then,

(*gen-stream #'simple-path*)

defines an infinite sequence of objects located at (*1,0,0*), (*2,0,0*), ... The objects in this stream have no model associated with them.

Object generators can be used as referents to transform motion or object streams that contain substantive objects. These functionally defined motions can be subsequently transformed by other object streams to prescribe a motion relative to the path of some object. For example, this could be used to define objects moving in orbits around other moving objects. In section 9, we present an example that uses this facility to create a camera that moves in circle around a

moving reference object.

## 8. Ports: Stream Visualization

It is critical that visual access to streams be provided. A stream of cameras is needed in order to view a motion stream. The intention is that successive scenes are to be projected onto successive members of the camera stream. The camera's model determines the projection function. This information is sufficient to map the scene onto a continuous, two dimensional image plane. The image plane must be then mapped onto a discrete display surface called a screen. Associated with the screen are state variables that determine the mapping from image space into screen space.

An object drawing function causes objects to be drawn on a screen with a projection function specified by a camera. To display a scene, the object drawing function must be mapped over all objects in the scene. However, the usual mapping function is inadequate for the purpose of drawing a scene on a screen. Because *stream-map* function uses delayed evaluation, only the first object will be immediately displayed. We must use a mapping function that forces evaluation of the entire stream to see the whole scene. The function *forced-map* causes immediate application of a function to all members of a stream in succession. To animate a motion stream, a nesting of *forced-maps* is required to draw all objects in all scenes.

## 9. Examples of Stream Editing

To illustrate the use of streams, we present two examples. The first example demonstrates the composition of several basic stream operations to form a useful higher order stream operation. Suppose a subset of the objects in a motion stream is to be selectively transformed without disruption of the other objects in the stream. The mapping function for motion streams, *mstream-geo-trans*, allows only transformation of entire streams. It would be useful to construct a new method of mapping transforms over a motion stream that would apply the transform only to those objects that satisfy a given predicate. We define a function

(*selective-geo-trans object-stream motion-stream pred*)

that will selectively transform only objects satisfying the predicate *pred*. This is accomplished in three steps. In the first step the motion stream is decomposed into two disjoint streams using *filter* and *nfilter*. One stream consists of objects satisfying *pred* that are to be transformed. The stream contains the remaining components of the input stream that are to remain unaltered.

The second step is to geometrically transform the former motion stream. The last step is to conjoin the transformed stream and the stream containing the remaining members of the original stream. The lisp definition of this function is:

```
(defun selective-geo-trans object-stream motion-stream pred)
  (stream-map #'merge
    (mstream-geo-trans object-stream (nfilter pred motion-stream))
    (filter pred object-stream)
  )
).
```

A schematic depiction of the function is shown in Figure 3.

The second example demonstrates the application of a functionally defined trajectory to create an object stream in which a camera moves in a circle orbiting another moving object. The camera's view vector is adjusted to be directed toward the object's center. We first use the camera object generator to create a single camera with a frame that matches that of the world coordinate system:

```
(setq cam (create-cam *base-frame*)).
```

The global variable *\*base-frame\** is the frame of the world coordinate system. Rather than limit ourselves to a particular circular path, we will allow the flexibility to specify the radius of the circles and the rate at which the camera moves along the curve. Let the function (*circular-path rad rate*) return a parametric function that defines frames along a circular path in the *X-Y* plane of radius *rad*. The argument *rate* determines the period of the cyclic motion. We first create a stream of reference objects that move along the circle:

```
(setq circ-stream (gen-stream (circular-path rate rad))).
```

To derive a stream of cameras moving along a circular path, we use the circular stream of

objects as referents to transform a stream of cameras:

```
(setq circ-cam-stream  
  (stream-map #'geo-trans-object circular-stream (stream-of cam))).
```

Now, we must transform the circular stream of cameras to move with reference to the object we wish to view. This is accomplished with another geometric transform using the circular camera stream as the target:

```
(setq trans-cam-stream  
  (stream-map #'geo-trans-object ref-stream circ-cam-stream)).
```

Finally, the camera must be oriented such that the view vector is always directed towards center of the reference object. An object transform function exists for this purpose. The function (*view-object ref target-cam*) returns a new camera object located at the position of the target camera with a view vector directed towards the object *ref*. This transform is mapped over the reference and camera streams to complete our task:

```
(setq final-cam-stream  
  (stream-map #'view-object ref-stream trans-cam-stream)).
```

A sequence of images from a camera circling a model of a walking robot are shown in Figure 4.

## 10. Summary

Motion stream editing allows a user to interactively choreograph intricate movement sequences. By treating the camera the same as any other moving object, viewing can be tied to the dynamics of the situation. Multiple views and scripted camera motion provide powerful tools for rendering animations of motion models.

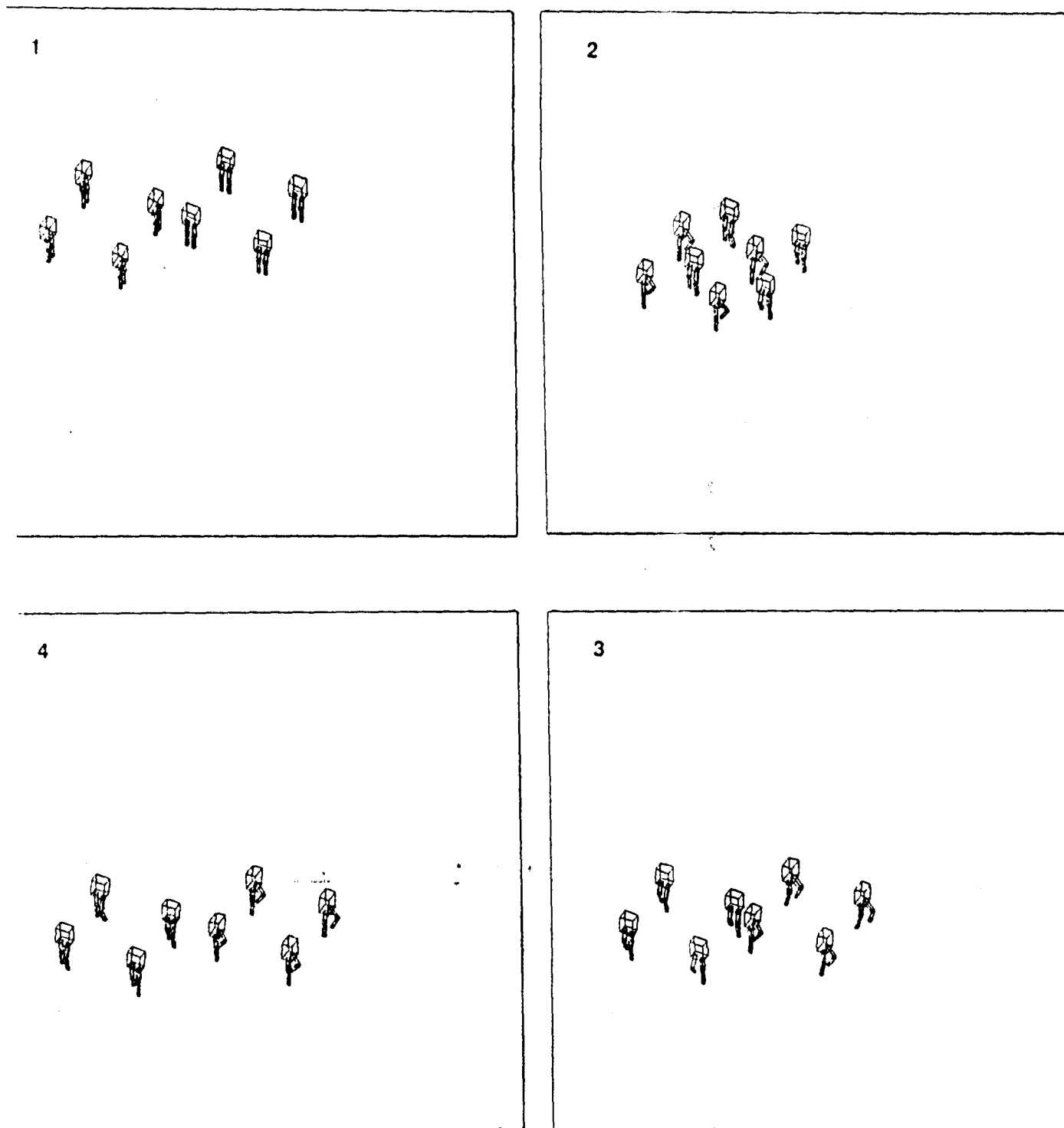
The system is extensible and easy to use for a user with modest training in Lisp. The focus of the presentation has been on the use of streams for editing the motion of bodies with rigid members and cameras. However, the approach could easily be adapted to include other types of objects that undergo parametric variation over time. Moving light sources with changing brightness or chroma could be added. Deformable objects which bend or grow could also be included. The techniques could be applied to models defined in an arbitrary space of any dimension.

Computer motion modeling frees animation from the notion of a sequence of two-dimensional movie frames. The animator can poke the camera in among the moving objects and direct the three-dimensional motion of the object actors with little effort and minimal cost. These techniques are valuable to present optimal visualization of dynamic phenomena. Moving, stereo cameras can be created as easily as single cameras to improve the perception of three-dimensionality. Motion stream editing provides the ability to design dynamic artificial environments that can define the circumstances for further simulation and testing.

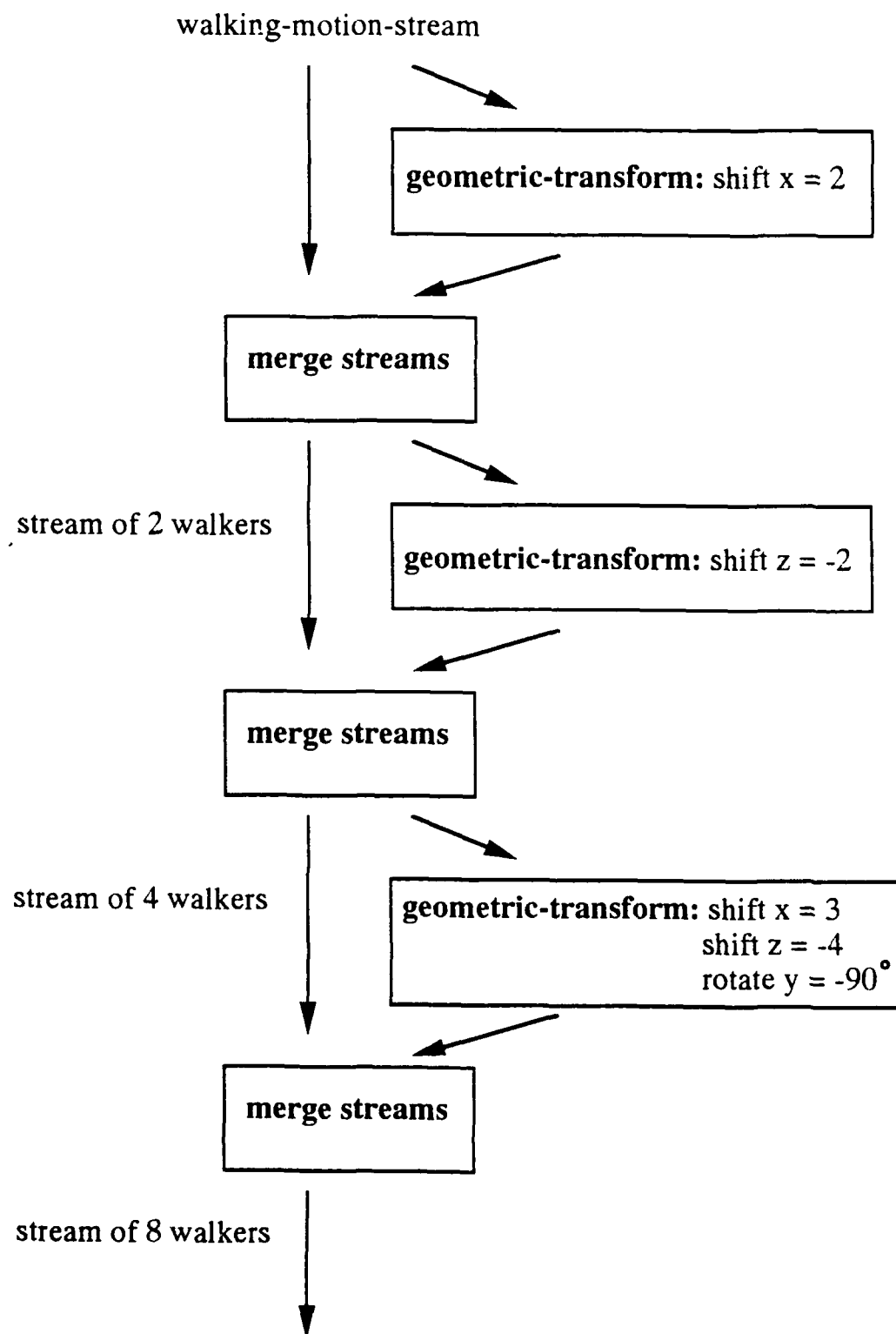
## References

1. H. Abelson and G. J. Sussman, in *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.
2. W. W. Armstrong and M. W. Green, "The dynamics of articulated rigid bodies for purposes of animation," *The Visual Computer*, vol. 1, pp. 231-240, 1985.
3. D. Bae and E.J. Haug, "A recursive formulation for constrained mechanical system dynamics: Part I, open loop systems," *to appear in Mechanics of Structures and Machines*.
4. D. Bae and E.J. Haug, "A recursive formulation for constrained mechanical system dynamics: Part II, closed loop systems," *to appear in Mechanics of Structures and Machines*.
5. J. Cremer, "An Architecture for General Purpose Physical Simulation -- Integrating Geometry, Dynamics, and Control," Ph.D. Thesis, TR 89-987, Cornell University, April, 1989.
6. C. M. Hoffmann and J. E. Hopcroft, "Simulation of physical systems from geometric models," *IEEE Journal of Robotics and Automation*, vol. RA-3, no. 3, pp. 194-206, June 1987.
7. G. L. Steele Jr., in *Common Lisp*, Digital Press, Bedford, MA, 1990.

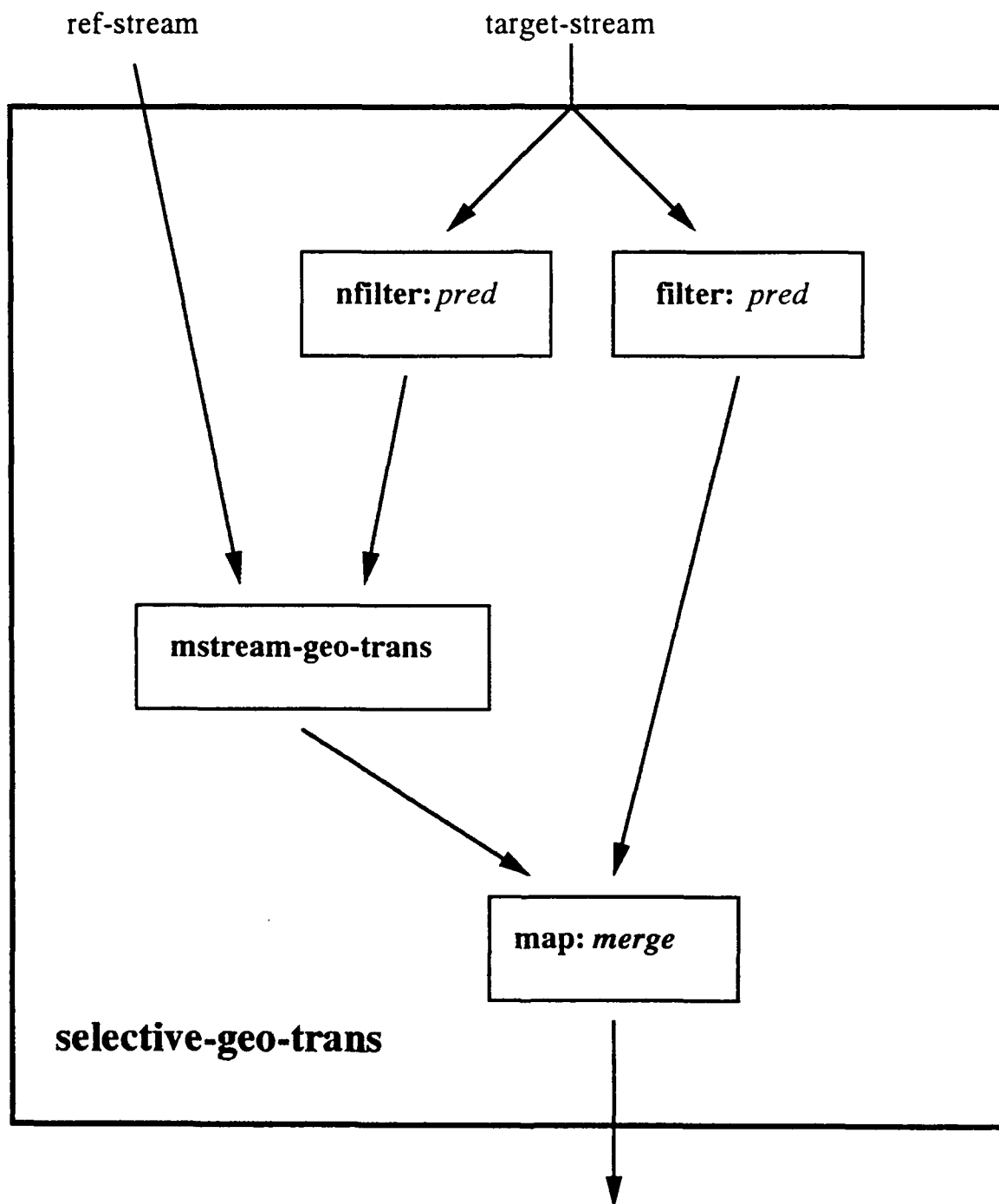
8. N. Magnenat-Thalmann and D. Thalmann, in *Computer Animation: Theory and Practice*, Springer-Verlag, Tokyo, Japan, 1985.
9. N. Magnenat-Thalmann and D. Thalmann, in *State-of-the-art in Computer Animation: Proceedings of Computer Animation '89*, Springer-Verlag, Tokyo, Japan, 1989.
10. A. J. Stewart and J. F. Cremer, "Algorithmic Control of Walking," *Proceedings of the IEEE International Conference on Robotics and Automation*, p. 1598, 1989.
11. Witkin and Cass, "Spacetime Constraints," *SIGGRAPH*, p. 159, 1988.



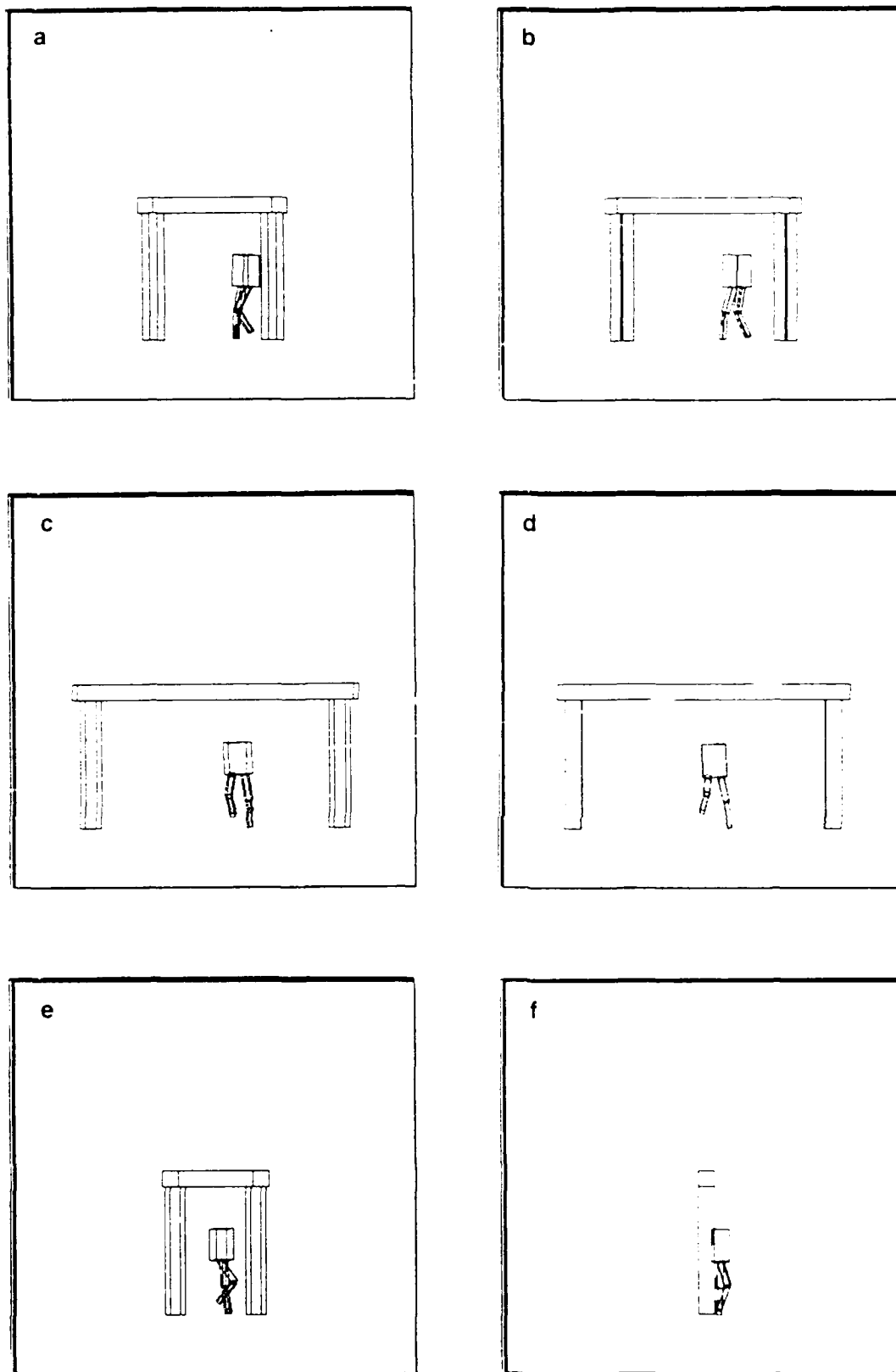
**Figure 1.** Selected scenes from a motion sequence with eight walking figure moving in groups of four. The scenes are ordered clockwise beginning with the upper, left panel. The sequence was created using the stream editing program diagrammed in figure 2.



**Figure 2.** A schematic representation of the stream editing program to create 8 marchers.



**Figure 3.** A schematic representation of a higher-order stream operation.



**Figure 4.** Scenes from a sequence viewed from a moving camera. The camera circles the walker with its view vector always directed towards the walker as the walker passes through a stationary arch.

- 83-10\* D. A. Eichmann, *A preprocessor approach to separate compilation in Ada.*
- 83-11\* R. K. Shultz, *Simulation of multiprocessor computer architectures using ACL.*
- 84-01 D. H. Freidel, *Modelling communication and synchronization in parallel programming languages.*
- 84-02\* T. Rus and F. B. Herr, *An algebraic directed compiler generator.*
- 84-03 M. E. Wagner, *Performance evaluation of abstract data type language implementations.*
- 84-04\* R. Ford and M. Wagner, *Performance evaluation methodologies for abstract data type implementation techniques.*
- 84-05 K. Brinck, *The expected performance of traversal algorithms in binary trees.*
- 84-06 K. Brinck, *On deletion in threaded binary trees.*
- 84-07 K. Siu-ming Yu, *A testbed database generator.*
- 84-08 D. Sawmiphakdi, *A multiprocess design for an integrated programming environment.*
- 84-09 D. W. Jones, *Machine independent SMAL: a symbolic macro assembly language.*
- 84-10\* R. K. Shultz, *Comparison of database operations on a multiprocessor computer architecture.*
- 84-11 J. H. Kingston, *A new proof of the Garsia-Wachs algorithm.*
- 85-01 T. Rus, *Fast pattern matching in strings.*
- 85-02 T. Rus, *An inductive approach for program evaluation.*
- 85-03 G. S. Singer, *Extensions to the Iowa logic specification language.*
- 85-04 A. C. Fleck, *Babble reference manual.*
- 85-05 K. Brinck, *Computing parent nodes in threaded binary trees.*
- 85-06 J. H. Kingston, *The amortized complexity of Henriksen's algorithm.*
- 85-07\* R. Ford, M. J. Jipping, and R. Shultz, *On the performance of an optimistic concurrent tree algorithm.*
- 85-08\* D. W. Jones, *Iowa capability architecture project ICAP programmer's preference manual.*
- 85-09\* S. P. Miller, *Automated instrumentation of communication protocols for testing and evaluation.*
- 86-01 M. J. Jipping, *An information-based methodology for the design of concurrent systems.*
- 86-02\* H. I. Mathkour, *An extended abstract data type specification mechanism.*
- 86-03 D. E. Glover, *Experimentation with an adaptive search strategy for solving a keyboard design/configuration problem.*
- 86-04 W. Pan, *Designing an operating system kernel based on concurrent garbage collection.*
- 86-05 B. C. Wenhardt, *A comparison of concurrency control algorithms for distributed data access.*
- 86-06 R. Shultz and I. Miller, *An execution cost analysis of multiple processor join methods.*
- 86-07 R. Shultz, *Controlling testbed database characteristics.*
- 86-08 R. E. Cantenbein, *Dynamic binding of separately compiled objects under program control.*
- 86-09 M. Pfreundschuh and R. Ford, *A model for modular system builds based on attribute grammars.*
- 86-10 R. Shultz and I. Miller, *Memory capacity in multiple processor joins.*
- 86-11\* M. P. Pfreundschuh, *A model for building modular systems based on attribute grammars.*
- 87-01\* M. J. Kean, *A communications architecture for distributed applications comprised of broadcasting sequential processes.*
- 87-02\* B. A. Julstrom, *A model of mental image generation and manipulation.*
- 87-03\* M. J. Jipping and R. Ford, *An information-based model for concurrency control.*
- 87-04\* Ravi Mukkamala, *Design of partially replicated distributed database systems: an integrated methodology.*
- 87-05\* A. C. Fleck, *A case study comparison of four declarative programming languages.*
- 88-01 Jing Jan, *Data abstraction in the Iowa logic specification language.*
- 88-02\* J. P. Le Peau and T. Rus, *Interactive parser construction.*
- 88-03\* J. Gilles, *A window oriented debugging environment for embedded real time ada systems.*
- 88-04 S. R. Sataluri and A. C. Fleck, *Incremental development of semantics using relational attribute grammars.*
- 88-05 S. R. Sataluri, *Generalizing semantic rules of attribute grammars using logic programs.*
- 88-06 Hantao Zhang, *Reduction, superposition and induction: Automated reasoning in an equational logic.*
- 89-01 C. M. Gessner, *A case study in post-developmental testing.*
- 89-02 Ken Slonneger, *Denotational semantics in prolog.*
- 89-03 Deepak Kapur and Hantao Zhang, *RRL: Rewrite rule laboratory user's manual.*
- 89-04 Hantao Zhang, *Prove ring commutativity problems by algebraic methods.*
- 89-05 David Allen Eichmann, *Polymorphic extensions to the relational model.*
- 89-06 In Jeong Chung, *Improved control strategy for parallel logic programming.*
- 89-07 Po-zung Chen, *New directions on stochastic timed petri nets.*
- 89-08 Frank William Miller, *A predictive real-time scheduling algorithm.*
- 90-01 Teodor Rus, *Algebraic construction of a compiler.*
- 90-02 Sukumar Ghosh, *Understanding self-stabilization in distributed systems, part I.*
- 90-03 Ching-Ming Chao, *A rapid prototyping methodology for conceptual database design using the executable semantic data model.*
- 90-04 H. S. Park, *Abstract object types = abstract data types + abstract knowledge types + abstract connector types.*
- 90-05 S. Amin and H. S. Park, *KEED: Knowledge engineering environment for diagnostic problems.*
- 90-06 H. S. Park, *Abstract knowledge prototyping.*
- 90-07 J. K. Kearney and S. Hansen, *Generalizing the hop: object-level programming for legged motion.*
- 90-08 J. K. Kearney and S. Hansen, *Stream editing for animation.*